Introduction

Common data structures, such as Lists, Stacks, Queues, Trees, Maps, etc., organise information in ways that constrain the operations which can be performed upon them, resulting in distinct performance characteristics. Moreover, the way in which these conceptual structures are implemented, may further affect their behaviour.

This assignment asks you to provide a very simple implementation of a List of Strings (one that uses Java's Object-Reference distinction to *explicitly* represent its successor-predecessor relationship). This time, **try to write sets of test data** ~~"input / expected-output" values~~ ***before*** **actually implementing your program**, then use them to confirm your new List class does indeed function correctly.

Finally, **reflect on the process**:

- Did it help you write the code correctly in the first place?
- How many mistakes did you catch after writing the code, that might otherwise have been missed?
- How many mistakes did you make when entering the test values by hand?
- How many mistakes did you make while manually checking the resulting outputs?
- How much time did it take to perform the tests?
- How much extra time did you spend redoing the tests after making a change
  (you did redo all the tests, didn't you?)

If you haven't already done so, perhaps it's time to **look at automated testing** using, for example, JUnit (which is built-in to DrJava and many other IDEs). Take a look at "Chapter 7. Testing Using JUnit" of the DrJava help ~**press F1 in DrJava**~ and try writing at least one or two such tests for your List class.

> ***Important:*** *DrJava includes an old, but usable version of JUnit (version 3.x, I think; however, the current stable version of JUnit is 5.4.x). You can configure DrJava to use other newer versions of JUnit, but for this lab assignment simply use whatever is already built-in.*

The List class

Implement the *List* class as specified below. Note that the *List* class must be in the "*cs102.ds*" package, must have *only* a single property, *head*, and that each *Node* must link *only* to the *next Node* in the list (i.e. it is a singly linked list, having no (explicit) link to the previous *Node*). You should implement the methods using iteration or recursion, as specified.

```
public class List
   public List()
   public void    addToHead( String item)
   public void    addToTail( String item)    // iterative method
   public String  removeFromHead()
   public boolean isEmpty()
   public String  getData( int index)        // invalid index returns null
   public void    print()                    // iterative method
   public void    printReverse()             // recursive method

   public boolean contains( String target)   // true if target is in the list, else false
   public boolean isOrdered()                // true if values strictly ascending, else false
```

Hint: *Use the Node class, below.*
Make it a private inner class of List... its properties can then be accessed directly!

```
private class Node {
   String data;
   Node next;
   public Node( String data, Node next) {
      this.data = data;
      this.next = next;
   }
} // end class Node
```

More hints? Try writing,

- a *toString()* method that returns a String representation of the list (in the same way ArrayList does).
- a private helper method *Node next(Node n)*,
  that returns a reference to *n*'s successor or null if it doesn't have one.
- a private helper method *Node previous(Node n)*,
  that returns a reference to *n*'s predecessor or null if it doesn't have one.
- a private helper method *Node tail()*, that returns a reference to the last Node in the list.

Creating and combining Lists

Add a public static method *List createFrom(String[])* that creates a new list initialised from the given array of Strings. You might also add another overloaded version that creates a List from a single String, such that each character in the String becomes a Node in the resulting list. Doing this should make it easier to test the following method...

Add the following method to your *List* class and demonstrate its correct functioning.

```
// Given two lists, a & b, return a new list that contains
// only those elements of a and b, that are not on both lists.
//
// For example, if a is { "A", "D", "C" } & b is { "K", "B", "A", "C", "R" }
// calling List.merger( a, b) might return { "D", "K", "B", "R" }
// or { "B", "D", "K", "R" } ~the order of the elements doesn't matter.
//
// You can assume the input lists do not themselves contain duplicate elements,
// that is, { "A", "D", "C" } is ok, but { "A", "D", "D" } is not
// since "D" appears twice.
//
public static List merger( List a, List b)
```

What is the (big-O) complexity of your method?

Could the same task be done more efficiently if the input lists were ordered?
